Spring History:

Spring is light weight and open source framework provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly.

Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003. Currently its maintained by pivotal.

Spring initial name is Interface 21. Later its modified to Spring.

Spring introduced alternative to EJB. EJB are used in heavy weight because its need AppServer as must to run the Application.

Spring is light weight because it wont depend on any Appserver and we can run the app with the help of spring library and jdk.

Spring is loosly Coupled. For example in india we have GSM(Ex Airtel if we are not interested in Airtel we can port it and move to any network it means loosely coupling.) and CDMA(Specific to that notwork only it means tightly coupiling)

Latest spring version in 2021 is 5.3.4.

New Features of Spring 5

****

JDK baseline update: We must require Java 8 and above version.

Core framework revision : @Nullable and @NotNull annotations will explicitly mark nullable arguments and return values. This enables dealing null values at compile time rather than throwing NullPointerExceptions at runtime.

Functional Programming With Kotlin

Testing Improvements: JUnit 5 Jupiter support started.

Library Support: JDBC 4, Hibernate 5 etc.

Framework:

Framework, is a platform for developing software applications.It provides a foundation on which software developers can build programs for a specific platform.

Web Frame Work: Its for only web application development. Ex: Struts,JSF

Application Frame Work: Its for developing the Web as well as enterprise apps. Ex: Spring

Advantages of Spring:

****

Loose Coupiling: The Spring applications are loosely coupled because of dependency injection.

Easy to test: The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

Lightweight: Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

Fast Development: The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

Powerful abstraction: It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

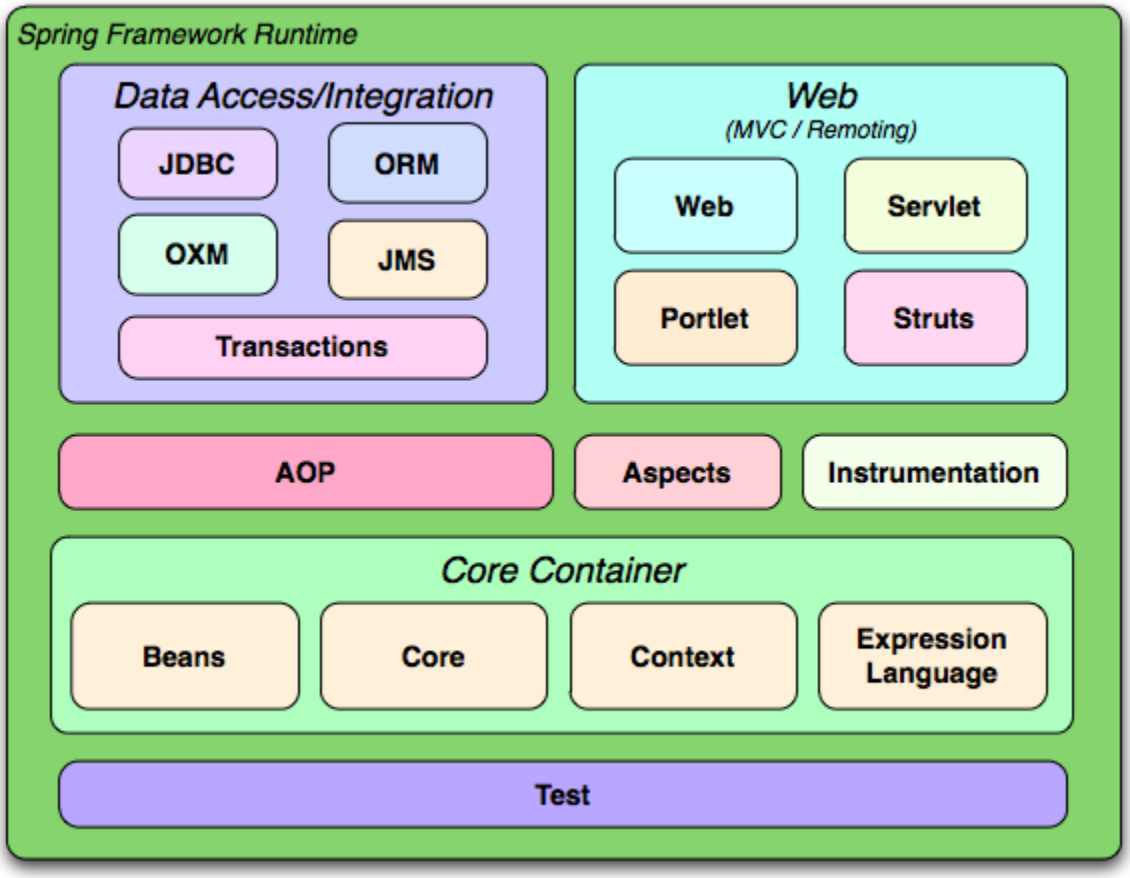
Easy integration: Easy to integrate any modules.

Predefined Templates: Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Spring Modules:

----

1. Core
2. Web
3. Test
4. AOP
5. Data Access/Integration



Spring Core Module:

---

i)This module is used to develop the Standard applications.

ii)This module will provide us a way of implementing the Dependency Injection (DI) in spring frame work.

iii)Spring core module provides a spring container.

iv)This core module will provides us the way of implementing spring bean life cycle operations.

Below things will come under Core Module:

****

1. Core

2. Bean

3. Context

4. Expression Language

Core

****

These modules provide IOC and Dependency Injection features.

Bean:

****

The Bean module provides BeanFactory, which is a sophisticated implementation of the factory pattern

Context

****

This module supports internationalization (I18N), EJB, JMS, Basic Remoting.

Expression Language

****

It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.

Spring Inversion of Control Container (IOC Container):

Spring IOC container is the core of the spring framework. Its is responsible for instantiating and manage the objects ans making them useful for required classes and its manage the entire life cycle of object from creation to destruction.

Spring container get the instructions like object instantiation, autowiring the objects by reading the consfiguration meta data provided from spring configuration xml/Annotations.

Types of IOC Container

==========

1.BeanFactory

2.ApplicationContext

Both BeanFactory and ApplicationContext provides a way to get a bean from Spring IOC container by calling getBean("bean name")

BeanFactory:

==========

This is the root interface for accessing the Spring container. To access the Spring container, we will be using Spring's dependency injection functionality using this BeanFactory interface

Usually, the implementations use lazy loading, which means that Beans are only instantiating when we directly calling them through the getBean() method.

Example:

```
XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("beans.xml"));

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

obj.getMessage();
```

ApplicationContext:

==========

The ApplicationContext is the central interface within a Spring application that is used for providing configuration information to the application.

It implements the BeanFactory interface. Hence, the ApplicationContext includes all functionality of the BeanFactory and much more! Its main function is to support the creation of big business applications.

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Exmple::

```
ApplicationContext context=new ClassPathXmlApplicationContext("beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();
```

BeanFactory VS ApplicationContext:

==========

The ApplicationContext includes all the functionality of the BeanFactory. It is generally recommended to use the former. There are some limited situations, such as in mobile applications, where memory consumption might be critical. In those scenarios, it would be justifiable to use the more lightweight BeanFactory. However, in most enterprise applications, the ApplicationContext is what you will want to use.

Dependency Injection:

****

Dependency Injection is a process of removing the dependency from programming code so that its easy to manage and test the application. In such case we provide the information from the external source such as XML file or Annotation. It makes our code loosely coupled and easier for testing.

Types of Dependency Injection:

==========

1.Setter Bases Dependency Injection

2.Constructor Based Injection

3.Field Injection

Constructor Based Injection:

==========

The process of injecting the dependencies through constructor is called as Constructor Based Injection.

We can insert the below values in Constructor Based Injection,

1.Primitive and String-based values

2.Dependent object (contained object)

3.Collection values etc.

Ex:

```
@Component
class Sandwich {
```

```
 private Topping toppings;
 private Bread breadType;
 private String name;
 private List<Address> addressList;

 Sandwich(Topping toppings) {
  this.toppings = toppings;
 }

 @Autowired
 Sandwich(Topping toppings, Bread breadType,name,addressList) {
  this.toppings = toppings;
  this.breadType = breadType;
        this.name = name;
        this.addressList = addressList;
 }
 ...
}
```

Setter Bases Dependency Injection

==========

The process of injecting the dependencies throgh setter method is called as Constructor Based Injection.

We can insert the below values in Constructor Based Injection,

1.Primitive and String-based values

2.Dependent object (contained object)

3.Collection values etc.

Exmple:

```
@Service
class DependentService {
   private final Service1 service1;
   private Service2 service2;
            private String name;
   private List<Address> addressList;

   @Autowired
   public DependentService(Service1 service1) {
      this.service1 = service1;
   }

            @Autowired
   public void Service1(Service2 service2) {
      this.service2 = service2;
   }

   @Autowired
   public void setName(SString name) {
      this.name = name;
   }

            @Autowired
   public void setAddressList(List<Address> addressList) {
      this.addressList = addressList;
   }
```

```
    void doSmth() {
        service1.doSmth();
        service2.doSmth();
    }
}
```

Field Injection:

==========

This type of injection is possible only in the annotation based approach due to the fact that it is not really a new injection type — under the hood, Spring uses reflection to set these values.

Advantages:

****

Easy to use, no constructors or setters required

Can be easily combined with the constructor and/or setter approach

Disadvantages

****

1. Less control over object instantiation. In order to instantiate the object of a class for a test, you will need either a Spring container configured or mock library — depends on the test you are writing.

2. A number of dependencies can reach dozens until you notice that something went wrong in your design.

3. No immutability — the same as for setter injection.

4. As there is no constructor/Setter methods it may create the same object twice.

Example:

****

```
@Service
class DependentService {
```

```
    @Autowired
    private Service1 service1;
    @Autowired
    private Service2 service2;

    void doSmth() {
        service1.doSmth();
        service2.doSmth();
    }
}
```

Spring Beans:

****

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container.

Beans Scopes:

==========

Singleton:

This scopes the bean definition to a single instance per Spring IoC container (default). WIll use this scope for state less beans.While using this scope, make sure bean doesn't have shared instance variables otherwise it might lead to data inconsistency issues.

prototype

This scopes a single bean definition to have any number of object instances.WIll use this scope for state full beans.

request

----

his is same as prototype scope, however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.

session

A new bean will be created for each HTTP session by the container.

global-session

This is used to create global session beans for Portlet applications.

Example:

<bean id = "..." class = "..." scope = "singleton"/>

<bean id = "..." class = "..." scope = "prototype"/>

<bean id = "..." class = "..." scope = "request"/>

<bean id = "..." class = "..." scope = "session"/>

<bean id = "..." class = "..." scope = "global-session"/>

Annotation based Spring Bean Scopes:

****

We can define the scope of a bean as prototype using scope="prototype" attribute of <bean/> element or using @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE) annotation.

Example:

```
@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class UserService {

  private String userName;
```

```
  public String getUserName() {
    return userName;
  }

  public void setUserName(String userName) {
    this.userName = userName;
  }
}
```

```
@Bean
@Scope(value="prototype")
public MyBean myBean() {
        return new MyBean();
}
```

Spring Beans Life Cycle:

==========

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed.

if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom init() method and the destroy() method.

Life Cycle Stages:

****

Container Started

Bean Inistantited

Dependencies Injected

Custom init() method

Custom utility method

Custom destroy() method

We can add the init and destroy methods in two ways:

XML Based:

```
<bean id="hw" class="beans.HelloWorld"
        init-method="init" destroy-method="destroy"/>
```

Coding based way:

----

we need to implement our bean with two interfaces namely InitializingBean, DisposableBean and will have to override afterPropertiesSet() and destroy() method. afterPropertiesSet() method is invoked as the container starts and the bean is instantiated whereas, the destroy() method is invoked just after container is closed.

Note: To invoke destroy method we have to call close() method of ConfigurableApplicationContext.

```
// HelloWorld class which implements the
// interfaces
public class HelloWorld
   implements InitializingBean,
 DisposableBean {

   @Override
   // It is the init() method
```

```
    // of our bean and it gets
    // invoked on bean instantiation
    public void afterPropertiesSet()
throws Exception
    {
        System.out.println(
            "Bean HelloWorld has been "
            + "instantiated and I'm the "
            + "init() method");
    }

    @Override
    // This method is invoked
    // just after the container
    // is closed
    public void destroy() throws Exception
    {
        System.out.println(
            "Conatiner has been closed "
            + "and I'm the destroy() method");
    }
}
```

Autowiring:

****

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

Autowiring Types:

****

1) no: It is the default autowiring mode. It means no autowiring bydefault.

2) byName: The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.

3) byType:The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.

4) constructor: The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

5) autodetect: It is deprecated since Spring 3.

Best way is autowire by properties, Setter Methods and Constructors

Advantages:

It requires the less code because we don't need to write the code to inject the dependency explicitly.

Disadvantages:

No control of programmer.

It can't be used for primitive and string values.